Быстрый старт

Здесь детально описан процесс создания простого приложения под Android с возможностями голосового управления с помощью API *Ассистента на русском*.

Что необходимо уметь

Для того, чтобы реализовать все нижеописанное, вам потребуются знания и навыки работы с языком программирования Java, использования Android SDK, и какая-либо IDE для разработки приложений под Android.

Что будет реализовано в этом примере

В этом примере мы реализуем простое приложение, умеющее знакомиться с пользователем, запоминать его имя и здороваться с ним каждый раз, обращаясь к нему по ранее сохранённому имени.

Эта функциональность может быть реализована в виде отдельного приложения под Android, либо встроена в уже имеющееся у вас приложение.

Все исходные коды примеров доступны на GitHub.

Создание нового проекта

В используемой IDE создайте новый проект приложения Android, либо используйте уже имеющийся. Нам потребуется создать несколько файлов исходных кодов и файлов с описанием грамматик запросов.

Мы предлагаем использовать последнюю версию Android Studio и Gradle. Для нового проекта вам потребуется прописать в файле `build.gradle` вашего проекта следующее

```
repositories {
    maven {
```

```
        url 'http://voiceassistant.mobi/m2/repository'
    }
}

dependencies {
    compile 'mobi.voiceassistant:client:0.1.0-SNAPSHOT'
    compile 'mobi.voiceassistant:base:0.1.0-SNAPSHOT'
}
```

Это настройка зависимостей от двух библиотек ассистента, которые будут использованы вашим приложением.

Обратите внимание на постфикс SNAPSHOT - он указывает, что данная версия библиотеки не финальная. В ближайшем будущем она будет часто обновляться

Создание грамматики запросов

Создайте файл hello.xml в директории xml вашего проекта. В нём мы опишем грамматику запросов, позволяющую преобразовать речь пользователя в команды вашему приложению.

> ### Как речь превращается в команды
>
> *Ассистент на русском* использует технологию распознавания речи, которая преобразует речь пользователя в текст. Из этого текста ассистенту необходимо "понять", к какому приложению относится этот текст, какая команда и в каком контексте должна быть выполнена, и выделить из этого текста данные, необходимые для выполнения команды.
>
> Для этого в грамматике запросов необходимо написать паттерны, которые опишут, на какие фразы пользователя нужно реагировать вашей программе, и какие данные из этой фразы необходимо получить для выполнения тех или иных команд.

В файле hello.xml напишите следующее

```
<?xml version="1.0" encoding="utf-8"?>

<module xmlns:android="http://schemas.android.com/apk/res/android">
```

```
    <command android:id="@+id/cmd_hello">
        <pattern value="привет* *"/>
    </command>

</module>
```

Здесь мы пока описали только одну команду с идентификатором `cmd_hello` и одним единственным паттерном. Этот паттерн будет срабатывать на фразах типа "Привет", "Приветы", "Привет как дела" и т.п., начинающиеся с формы слова *привет* с любым окончанием и заканчивающиеся любым количеством любых других слов.

Создание агента

Агент - это программный интерфейс между ассистентом и бизнес-логикой вашего приложения. По сути это надстройка над стандартными Android-сервисами, содержащая специальные методы по управлению диалогом с пользователем.

Для создания агента нам потребуется написать класс, реализующий абстракцию `AssistantAgent`

```
public class HelloAgent extends AssistantAgent {
    @Override
    protected void onCommand(Request request) {
    }
}
```

Каждый агент обязан переопределить по крайней мере один метод - `onCommand`, который является точкой входа для ассистента. В этот метод ассистент передаёт специальный контейнер данных `Request`, содержащий всё необходимое для обработки запроса от пользователя.

```
@Override
protected void onCommand(Request request) {
    switch (request.getDispatchId()) {
        case R.id.cmd_hello:
            onHello(request);
            break;
    }
```

```
}

private void onHello(Request request) {
}
```

Здесь мы диспетчеризуем нашу пока единственную команду с идентификатором `cmd_hello` в метод `onHello`, где впоследствии реализуем логику приветствия и знакомства с пользователем. Для этого мы пользуемся методом `getDispatchId` класса `Request`, который возвращает целочисленный идентификатор команды.

> Все xml-файлы в андроиде компилируются, что позволяет использовать класс `R` для хранения уникальных целочисленных идентификаторов команд.

Генерация ответа

Агент обязан вернуть хотя бы один ответ на каждый запрос пользователя. Для этого необходимо создать контент ответа и добавить его в запрос методами `addResponse` или `addQuickResponse`.

> Ассистент ведёт историю запросов пользователя. Поэтому каждый ответ нужно связывать с запросом, что и обуславливает использование методов класса `Request` для генерации ответа пользователю.

Сейчас мы создадим простейший ответ, в котором пока просто поздороваемся. Это будет обычная строка текста со словом "Привет". Её лучше всего записать в ресурсном файле `values/strings.xml`.

```
<resources>
    <string name="hello_hello">Привет</string>
</resources>
```

Тогда метод `onHello` можно реализовать следующим образом

```
private void onHello(Request request) {
    request.addQuickResponse(getString(R.string.hello_hello));
}
```

Здесь мы используем метод `addQuickResponse` для формирования быстрого ответа, передавая в качестве контента простую строку текста. Этот текст будет отображён в интерфейсе ассистента в виде "бабла" с текстом, и ассистент проговорит этот текст, если у пользователя включён TTS.

Регистрация агента и запуск приложения

Для того, чтобы ассистент смог обнаружить агента и загрузить его основной модуль, необходимо описать его в манифесте вашего приложения AndroidManifest.xml

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.quickstart"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="18" />

    <application android:label="@string/app_name"
        android:icon="@drawable/ic_launcher">

        <service android:name=".HelloAgent">

            <intent-filter>
                <action android:name="mobi.voiceassistant.intent.action.COMMAND"/>
                <data android:scheme="assist" android:host="mobi.voiceassistant.ru"/>
            </intent-filter>

            <meta-data android:name="mobi.voiceassistant.MODULE" android:resource="@xml/hello"/>

        </service>
    </application>

</manifest>
```

Как видим, агент является обыкновенным сервисом. В метаданном `mobi.voiceassistant.MODULE` указывается основной модуль, который

будет загружен ассистентом автоматически. Также необходимо описать action `mobi.voiceassistant.intent.action.COMMAND` в фильтре агента, указав в тэге `data` с каким из приложений "Ассистента на русском" работает ваш агент. В данном случае это русская версия ассистента (пакет `mobi.voiceassistant.ru`). Подробнее про регистрацию агента читайте в специальном разделе.

Теперь наше приложение можно собрать стандартными методами сборки Android-приложений и запустить на устройстве, где уже установлен "Ассистент на русском".

При запуске "Ассистент на русском" обнаружит на устройстве наше приложение и загрузит его основной модуль. После чего все подходящие фразы будут диспетчеризоваться к нашему агенту `HelloAgent` и пользователь будет видеть в ответ бабл с текстом "Привет".

> Как вы видите, наше тестовое приложение не содержит ни одной активности и не отображается в меню. Это не значит, что вы не можете использовать активности. Просто в нашем случае в этом нет необходимости.

После первого запроса вы можете подключиться стандартным дебагером к процессу вашего приложения.

Управление диалогом

Сейчас мы реализуем более сложную логику, в которой будем переключать контекст диалога, сохранять имя пользователя и здороваться с ним по имени.

Когда пользователь говорит "Привет", а наше приложение ещё не знает его имени, необходимо попросить пользователя представиться. Всё что пользователь скажет в ответ на этот вопрос будет интерпретироваться как имя. После получения имени приложение сможет сохранить его в какое-либо хранилище данных (в нашем случае мы будем использовать SharedPreferences).

Модальный режим

Для реализации такого сценария нам понадобится управлять диалогом, а именно входить в модальный режим. Подробнее о режимах диалога и их использовании читайте в специальном разделе. Пока что нам достаточно знать, что модальный режим позволяет ограничить набор доступных пользователю команд, как бы скрыв все остальные команды на время. Это нужно для того, чтобы перехватить нашим агентом любую строку и интерпретировать её как имя пользователя (потому что именя бывают разные и описать все имена в паттерне не представляется возможным).

Создайте ещё один xml-файл в директории xml вашего приложения с именем `name.xml` с таким содержимым

```
<?xml version="1.0" encoding="utf-8"?>

<module xmlns:android="http://schemas.android.com/apk/res/android">

    <pattern name="UserName" value="*"/>

    <command android:id="@+id/cmd_name">
        <pattern value="[меня зовут] $UserName"/>
    </command>

</module>
```

Здесь мы описали простой паттерн `UserName`, который "поймает" любую строку любой длины, и одну команду с паттерном фразы, которая *может* начинаться со слов "Меня зовут" и заканчиваться именем пользователя.

> Квадратные скобки обозначают, что слова "Меня зовут" являются необязательными - т.е. могут отсутствовать в тексте. Другими ловами, пользователь может сразу сказать своё имя. И тогда паттерн тоже сработает.

Теперь в агенте `HelloAgent` мы можем переключить контекст беседы в модальный режим с одним этим модулем, когда захотим узнать имя. Пользователь будет вынужден сказать своё имя, если не использует слово "Отмена" для того чтобы не отвечать на этот вопрос.

```
private void onHello(Request request) {
    final SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);
    final String userName = preferences.getString(PREF_NAME, null);

    if(userName == null) {
        final Response response = request.createResponse();
        response.setContent(getString(R.string.hello_say_name));
        response.enterModalQuestionScope(R.xml.name);
        request.addResponse(response);
    } else {
        request.addQuickResponse(getString(R.string.hello_hello, userName));
```

```
    }
}
```

Теперь при приветствии мы сперва заглядываем в SharedPreferences и пытаемся получить оттуда ранее сохранённое имя пользователя. Если мы его находим, то просто здороваемся с пользователем по имени. А если нет - то переводим диалог в модальный режим и задаём пользователю вопрос.

Файл `values/strings.xml` будет выглядеть следующим образом

```
<resources>
    <string name="app_name">AssistantQuickStart</string>
    <string name="hello_hello">Привет, %1$s!</string>
    <string name="hello_say_name">Привет. А как тебя зовут?</string>
</resources>
```

> Метод `enterModalQuestionScope` не просто заставит ассистента перейти в модальный режим, но ещё и автоматически включит микрофон после того, как ассистент озвучит текст вопроса.

Пользователь теперь должен либо назвать своё имя, либо сказать "Отмена" для выхода из модального режима. Иначе любая его фраза будет восприниматься нашим агентом как имя, т.к. сработает команда с идентификатором `cmd_name`.

Обработка запроса в модальном режиме

Ответ в модальном режиме почти ничем не отличается от ответа в обчном за исключением того, что пользователь может выйти из него с помощью команды "Отмена" или ответить что-то, что не может быть обработано паттернами указанного модуля. Если второй вариант нас не беспокоит в данном случае (т.к. любая фраза будет интерпретирована как имя), то на отмену нужно как-то прореагировать.

Для этого агент должен переопределить метод `onModalCancel`, в котором нужно тоже вернуть какой-либо контент для адекватного ответа пользователю. В нашем случае мы вернём простую строку с текстом "Пока"

```
@Override
protected void onModalCancel(Request request) {
    request.addQuickResponse(getString(R.string.hello_cancel));
```

```
}
```

В случае же нормального пользовательского ответа мы получим обычную команду, которую сможем обработать в методе `onCommand`

```
@Override
protected void onCommand(Request request) {
    switch (request.getDispatchId()) {
        case R.id.cmd_hello:
            onHello(request);
            break;
        case R.id.cmd_name:
            onName(request);
            break;
    }
}

private void onName(Request request) {
}
```

В методе `onName` мы сможем получить из запроса имя и сохранить в SharedPreferences, после чего поздороваться с пользователем по этому имени

```
private void onName(Request request) {
    final Token token = request.getContent();
    final Token nameToken = token.findTokenByName(TOKEN_NAME);
    final String userName = nameToken.getSource();

    if(userName.length() == 0) {
        final Response response = request.createResponse();
        response.setContent(getString(R.string.hello_say_again));
        response.enterModalQuestionScope(R.xml.name);
        request.addResponse(response);
        return;
    }
```

```
    final StringBuilder sb = new StringBuilder(userName);
    sb.setCharAt(0, Character.toUpperCase(userName.charAt(0)));
    final String displayName = sb.toString();

    final SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);
    preferences.edit().putString(PREF_NAME, displayName).commit();

    onHello(request);
}
```

В данном случае мы имеем дело с речевым взаимодействием с пользователем, поэтому контентом запроса будет являться токен - семантическое дерево разбора фразы. Этот токен содержит дочерний токен с именем UserName, который мы можем получить с помощью метода `findTokenByName`.

Токен UserName содержит простой текст в поле `source`, который мы можем интерпретировать как имя. Т.к. мы определили паттерн UserName как `*`, то это значит, что строка текста может быть и нулевой длины (например, если пользователь скажет только начало фразы "Меня зовут"). Для обработки этого случая мы проверяем длину строки токена и снова переводим ассистента в модальный режим с просьбой повторить имя если строка оказалась пустой.

Если же пользователь сказал имя, то мы можем сохранить его в SharedPreferences и сгенерировать ответ.

> Важно запомнить, что ассистент не переходит снова в модальный режим после отработки команды или метода `onMoalFail`.

Файл `values/strings.xml` будет содержать следующие строки

```
<resources>
    <string name="app_name">AssistantQuickStart</string>
    <string name="hello_hello">Привет, %1$s!</string>
    <string name="hello_say_name">Привет. А как тебя зовут?</string>
    <string name="hello_say_again">Прости, не понимаю. Как тебя зовут?</string>
    <string name="hello_cancel">Пока</string>
</resources>
```

Теперь можно снова собрать приложение и установить его на девайсе. После этого "Ассистент на русском" перезагрузит модуль и им можно будет сразу пользоваться.

Нестандартная озвучка ответа

Как вы можете заметить, TTS ассистента пытается использовать интонацию, подходящую для того текста, который мы вернули в ответе. Но не всегда эта интонация подходит в конкретном случае. Например, в нашем приложении синтаксически верно отделить слово "Привет" от имени пользователя запятой. Но при произношении приветствия в реальной жизни мы не ставим паузу между "Привет" и именем.

Поэтому нужно описать другую (более правльную) озвучку для приветствия, в которой не будет паузы. Для этого запишем в файле `values/strings.xml` ещё одну строку для озвучки

```
<string name="speech_hello">привет %1$s</string>
```

В ней мы убрали запятую между словами. Эта строчка будет озвучиваться ассистентом более правильно.

Для использования такого произношения нам понадобится использовать утилитный класс API `SpeechTextUtils`

```
private void onHello(Request request) {
    final SharedPreferences preferences = PreferenceManager.getDefaultSharedPreferences(this);
    final String userName = preferences.getString(PREF_NAME, null);

    if(userName == null) {
        final Response response = request.createResponse();
        response.setContent(getString(R.string.hello_say_name));
        response.enterModalQuestionScope(R.xml.name);
        request.addResponse(response);
    } else {
        final CharSequence content = SpeechTextUtils.textWithSpeech(getString(R.string.hello_hello, userName), getString(R.string.spe
        request.addQuickResponse(content);
    }
}
```

Здесь мы генерируем ответ в виде CharSequence, в котором текст для бабла и текст для озвучки различны.

Как поменять своё имя

Т.к. наше приложение хранит имя пользователя в SharedPreferences, то для смены имени можно просто очистить данные приложения через менеджер приложений. Но это нам не оченб подходит, и мы попробуем изменить грамматику нашего модуля так, чтобы пользователь в любой момент мог представиться ассистенту заново. Для этого можно изменить основной модуль hello.xml следующим образом

```
<?xml version="1.0" encoding="utf-8"?>

<module xmlns:android="http://schemas.android.com/apk/res/android">

    <pattern name="UserName" value="*" />

    <command android:id="@+id/cmd_hello">
        <pattern value="привет* *"/>
    </command>

    <command android:id="@+id/cmd_name">
        <pattern value="* меня зовут $UserName"/>
    </command>

</module>
```

Здесь мы всего лишь добавили команду `cmd_name` и паттерн `UserName` в наш основной модуль, после чего агент сможет обрабатывать фразу типа "Привет меня зовут Пётр" без входа в модальный режим и дополнительного вопроса. Агент просто перезапишет ранее сохранённое имя и поздоровается. Код менять не нужно.

> Как вы заметили, паттерн команды в основном модуле отличается от той же команды в модуле name.xml. В новом паттерне слова "Меня зовут" стали обязательными, т.к. без этого ваш агент реагировал бы на любую фразу, которая не подошла ни под один из паттернов других агентов.

В заключение

В этом примере мы реализовали простое приложение, которое подключается к ассистенту и устанавливает речевой контакт с пользователем. Мы научились менять контекст диалога, генерировать ответы (в том числе с различной озвучкой) и получать данные из речи пользователя.

Как видно из примера, приложение ничем не отличается от стандартного приложения под Android. В нём могут использоваться активности, сервисы, можно управлять сохранением данных и вообще проделывать все те операции, которые доступны любому Android-приложению.

> В данном примере мы использовали самый простой вид коммуникации с пользователем и самый простой GUI для отображения данных. Для создания гораздо более сложного GUI изучите раздел бабблы.

Далее мы предлагаем вам изучить тему "Архитектура", которая детально описывает каждый компонент ассистента и способы коммуникации с пользователем. Понимание этих аспектов позволит реализовать гораздо более сложные голосовые приложения из реальной жизни.